Injecting Ilities

Robert E. Filman*

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759-6509
filman@mcc.com

This paper discusses the use of aspect-oriented programming technology to impose desirable system-wide properties on distributed systems.

Overcoming Complexity in Distributed Computing

The goal of the Microelectronics and Computer Technology Corporation's (MCC) Object Infrastructure Project (OIP) is to simplify the development and evolution of distributed, object-oriented applications. OIP is designing and implementing an architecture where ility-providing program elements (*injectors*) can be automatically wrapped around application components. The task of specifying the appropriate injectors is separate from the actual component coding.

Traditionally, software application development has been a monolithic process. An organization building a software system presumed to know how it wanted that system to behave. The requirements for that behavior would flow down to the construction of the underlying modules. Since the modules were being built specifically for the system in question, it was "straightforward" to get their developers to obey the rules and conform to defined standards. To the extent that the system used an externally provided component such as a GUI or database, the behavior of that component would be ascertained and the use of that component within the architecture of the system shaped to match the external component's actual behavior.

Software development has gotten more complex. Technologies such as CORBA and HTTP provide the glue for building applications from distributed components. But understanding the nuances of multiple components and varieties of glue can is itself an intellectual challenge. We can't expect a single application programmer to become expert in the intricacies of many components, even if the application needs to use them all. Similarly, components impose their own constraints on their usage. We want to develop systems from components but don't want the artifacts of a particular component manufacturer to permeate our designs, rendering us eternally dependent on the whims, demands and destiny of that vendor. We want components that obey our policies—not to have to distort our systems to match the policies of the components. And we want ways to federate existing systems while still maintaining overarching rules and procedures.

---

* This paper describes work performed at MCC while the author was on assignment from Lockheed Martin Missiles and Space. LMMS contact information: Advanced Technology Center; Lockheed Martin Missiles and Space; 3251 Hanover Street O/H1-43 B/255; Palo Alto, California 94304. Email: bob.filman@lmco.com.

Distributed systems introduce additional complexity. Developing a distributed system is in itself a more difficult task because distributed systems imply non-deterministism(and non-determinism is complex), distribution introduces many additional kinds of failures, distribution is naturally less secure, and distribution's inherent decentralization is inconvenient to manage. Distributed computing can be made simpler by making it look more like conventional programming and by providing and automatically invoking correct implementations of distributed and concurrent algorithms.

Requirements

System development (ought to) follow from requirements. What kinds of requirements are there? In [1], I proposed dividing requirements into four classes based on the "tractability" of achieving the requirement: *functional* requirements that exhibit the primary semantic behavior of a system and are typically locally realized, *systematic* requirements that can be achieved by "doing the right thing" consistently throughout the program, *combinatoric* requirements that are computationally intractable expressions of overall system behavior (for example, guarantees of real-time behavior or limits on storage footprint) and *aesthetic* requirements that express non-computable qualities of the system (about which even wise men may differ.) The first of these is well supported by the conventional development process ("The requirements say that a dialog box appears at this time with the choices…") and the last two are difficult to automate in any case.

Aspect-oriented programming gives us a handle on systematic requirements. Always doing the right thing is hard for a single coder, no less a gaggle of hackers. Computers, on the other hand, can be programmed to be tediously consistent. (Compilers are programs that are tediously consistent in transforming other programs.)

What kinds of systematic requirements exist? Our applications should exhibit reliability, security, scalability, extensibility, manageability, maintainability, interoperability, composability, evolvability, survivability, affordability, understandability, and agility. (I've omitted a few.) Let us label these qualities *ilities*. The keen reader is likely to ask, "So what exactly do you mean by, say, reliability?" Reliability requirements differ for different applications and *are likely to change over the lifetime of the application*. The reliability requirement maps to executing specific algorithms that need to be invoked systematically throughout the application. Some ilities are consequently manifestations of properly defined and implemented requirements. The open scientific question is thus (given the fuzzy definition of an ility) "Which ilities can be achieved by systematic actions, and what are the actions needed to achieve those ilities?"

Controlling communication

The OIP project is pursuing the thesis that certain interesting ilities (security, reliability, manageability, quality of service) can be achieved by proper manipulation of the communications between components and the significant events of a component's lifecycle. We are currently creating a set of tools to realize the transformation from specified ilities to controlled communications, a *reference architecture* (set of rules defining component interactions) and set of *frameworks* (realizations of that architecture in particular environments) to demonstrate this thesis. A key observation of this work is that communication is not confined to the "actual text of a message" (for example, the procedure be-

ing called and its arguments) but also allows arbitrary additional annotation—we presume to control both sides of the communication act.

Our efforts are aspect-oriented programming [2] in that we are separating the tasks of creating the actual domain application from the code that produces security, reliability, and such ilities. Our efforts can also be seen as an instance of the perpetual effort in computer science to raise the "level" of supporting substrates. Tools such as CORBA have enabled programmers to code to the specification of objects and methods. But realizing elements such as security or reliability are still the responsibility of the application programmer, and likely to be done incorrectly or incompletely by most such programmers. (A programmer expert in the workings of a satellite flight control system or medical database is unlikely to also be expert in security and replication algorithms.) This effort can thus be seen as a way to produce the "next generation" of CORBA-like systems [3], where the application programmer no more worries about how to achieve security than she does about mapping the location of a mouse click to a window's button.

Applied ilities

Let us consider, for each of our target ilities, how communication and lifecycle control can be used to affect or realize that ility, and the limits of that realization.

Security

Security (at least in a software sense) is primarily a combination of access control, intrusion detection, authentication, and encryption. Controlling the communication process allows us to encrypt communications, reliably send user authentication from client to server (and pass it along to dependent requests) and check the access rights of requests, all independent of the actual application code. (However, depending on where the encryption happens in the communication process, we may only be able to encrypt the message data, not its headers.) Watching communications provides a locus for detecting intrusion events [4] (though not, of course, specifying the actual algorithms for recognizing an intrusion.) These mechanisms can all be imposed on a component-based system by controlling its communications. (Such mechanisms cannot, however, prevent subverting a system's personnel, tapping communication lines, brute-force cracking of encryption codes, or components that cheat by opening their own socket connections.)

Manageability

The International Standards Organization has defined five elements to manageability: performance measurement, accounting, failure analysis, intrusion detection, and configuration management. The first four of these can be implemented by generating events in relevant circumstances and directing those events to the appropriate recipients. To the extent that the semantics of these events can be tied to communication acts (e.g., each time a service is called, a micro-payment for that service is processed, or the trace of inter-component messages is sent to a system's debugger) then they can be realized through external communication controls. To the extent that the interesting actions happen completely within the application components (e.g., payment is due proportional to the number of records accessed by a database service or debugging wholly within a component) then this technique will prove inadequate.

Configuration management is partially an issue of object lifecycle. Communication control can be used to dynamically determine if appropriate configurations are in use and to automatically update stale configurations.

Reliability

Our primary experiments in supporting reliability have centered on using replication for reliability [5]. Replication algorithms typically need to send copies of messages to replicants, but our work has also revealed that message replication is insufficient for practical replication. Rather, the application needs to be written to express its operations in symbolic terms, not in terms of addresses in a specific replicant's address space.

Similarly, I believe transaction management would (practically) yield to communication control only if the managed objects provide the necessary primitives (locking and rollback.) These points illustrate the limitations of pure communication control in the presence of monolithic compents, even given the existence well-defined algorithms.

Quality of service

By quality of service I mean to encompass a variety of requirements for getting things done within time constraints. The real-time community recognizes two varieties of real-time systems, *hard real-time* and *soft real-time*. Hard real-time systems have tasks that must be completed at particular deadlines, or else the system is incorrect. Soft real-time systems seek to allocate resources so as to accomplish the most important things. To achieve hard real-time systems, one can either reserve resources and plan consumption or use an anytime algorithm. Aside from that latter, somewhat esoteric choice, hard real-time requires cooperation throughout the processing chain (for example, in the underlying network), for the promise of particular service can be abrogated in too many places. That is, you can't get hard real-time unless you build your entire system with that in mind. It's a combinatoric requirement. (Doug Schmidt's work on Real-Time CORBA ORBs [6] illustrates this point: commercial ORBs, built without constant real-time mindfulness, conceal FIFO queues and exhibit anti–real-time behavior.)

Soft real-time quality of service is amenable to several communication control tactics. These include calling the underlying system's quality of service primitives, using side-door mechanisms to efficiently transport large quantities of data (e.g., opening a socket to send a movie, thereby avoiding CORBA coding and decoding), using queue control to identify the most worthwhile thing to do next [7] and by choosing among multiple ways of problem solving. All of these except the last are well within the scope of communication control, and if the application supplies the alternative problem solving methods (either by replicating the problem solving sites or providing genuinely different algorithms) the communication control mechanism can learn (based on historical timing data and communications with other clients) the most efficient problem solvers.

Implementation

A few remarks on OIP implementation are worth mentioning. The underlying computational model is to wrap components with a sequence of injectors. Systems like CORBA and Java RMI support a stub/skeleton proxy mechanism for distributed communication. (In fact, several commercial ORBs include the ability to specify some user-defined filters on communications. Such filters are required by the yet-to-become-commercially-

available CORBA security service.) OIP injectors are individuated by proxy/method and can, with the appropriate access controls, be changed for a particular proxy.

Injectors can both read and write not only the application program arguments, but also the annotations associated with the message. Thus, the annotation mechanism supports communications among injectors.

Annotations can be understood to be the procedure-call analog of mail headers. Certain headers have common meanings ("From" and "To") while others are more specialized to particular programs ("X-Sun-Charset"). Our experience with OIP has suggested an initial set of common annotations (including session identification, request priority, sending and due dates, version and configuration, answer futures, cyber wallet, public key, sender identification and conversational thread.) Thus, a request can be identified as having a specific priority and injectors can change their actions based on that priority (for example, to queue incoming requests and execute the highest priority request next, as was done in [7].)

Security concerns about the use of injectors can be reduced by requiring an injector to declare which annotations it reads and writes (and enforcing that declaration). We may be more willing to use a plug-in injector obtained from a random site if we know that all it does is read the sending and due dates of messages than if it claims to alter sender identification and message text. (The former could be used to support an adaptive real-time mechanism such as the one described in [8].)

OIP also supports chaining of annotations through called threads. Thus a routine called with a priority of *X* will make calls at priority *X* (unless the thread explicitly changes the priority).

Selection of which initial injectors to use for which methods of which classes is done by a compiler that takes a language of injector specifications and builds the appropriate default structures for the run-time system. This language provides a level of indirection between desired ilities and their implementations, and allows the successive refinement of policies through an organization. This use of a separate specification language for creating filters parallels the work at BBN on QoS [9], where an IDL-like Quality Description Language is woven with IDL to affect system performance.

I also note that Videira Lopes and Kiczales also apply communication control to distribution for the aspects of synchronization and distribution [10].

Concluding remarks

I have argued that high-level, desirable system-level properties can be achieved in a component-based system by systematically controlling the inter-component communications and component lifecycle. Our initial experiments have lent credence to this hypothesis, subject to the caveats that some algorithms (e.g., transactions) require cooperation on the part of the application, and that our desire for system-level properties (e.g., security) must be kept within the range of definable mechanisms. Our work continues on developing the mechanisms to automate this process and testing our thesis.

Acknowledgments

The ideas expressed in this paper have emerged from the work of the MCC Object Infrastructure Project, particularly Stu Barrett, Carol Burt, Deborah Cobb, Tw Cook, Phillip Foster, Diana Lee, Barry Leiner, Ted Linden, David Milgram, Gabor Seymour, Doug Stuart and Craig Thompson. Some of these ideas have been expressed in reference [1].

My thanks to Tw Cook, Diana Lee, Ted Linden, Dave Milgram and Tom Shields for comments on the drafts of this paper.

References

[1] Robert E. Filman, "Achieving Ilities," *Workshop on Compositional Software Architectures*, Monterey, California, Jan. 1998.
http://www.objs.com/workshops/ws9801/papers/paper046.doc

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin "Aspect-Oriented Programming, " Xerox *PARC Technical Report, February 97, SPL97-008 P9710042.*
http://www.parc.xerox.com/spl/projects/aop/tr-aop.htm

[3] Craig Thompson, Ted Linden and Bob Filman, "Thoughts on OMA-NG: The Next Generation Object Management Architecture," Presented at the OMG Technical Meeting, Dublin, Ireland, September, 1997.
http://www.mcc.com/projects/oip/next_oma.html

[4] Robert Filman and Ted Linden, "Communicating Security Agents," The Fifth IEEE-Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises---International Workshop on Enterprise Security, Stanford, California, June 1996, pp. 86-91.

[5] Stu Barrett and Phillip Foster, "Turning Java Components into CORBA Components with Replication," OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, California, Jan. 1998.
http://www.objs.com/workshops/ws9801/papers/paper067.doc

[6] Douglas C. Schmidt, Rajeev Bector, David L. Levine, Sumedh Mungee, and Gurur Parulkar, "An ORB Endsystem Architecture for Statically Scheduled Real-time Applications," *Proc. IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, San Francisco, Dec. 1997, pp. 52-60.

[7] Diana Lee and Robert Filman, "Verification of Compositional Software Architectures," OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, California, Jan. 1998.
http://www.objs.com/workshops/ws9801/papers/paper096.html

[8] M. Gergeleit, E. Nett, and M. Mock, "Supporting Adaptive Real-Time Behavior in CORBA," *Proc. IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, San Francisco, Dec. 1997, pp. 61-67.

[9] Richard Schantz, David Bakken, David Karr, Joseph Loyall, and John Zinky, "Distributed Objects with Quality of Service: An Organizing Architecture for Integrated System Properties," OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, California, Jan. 1998.
http://www.objs.com/workshops/ws9801/papers/paper099.doc

[10] Cristina Videira Lopes, and Gregor Kiczales, "D: A Language Framework For Distributed Programming," Xerox PARC Technical report, February 97, SPL97-010 P9710047.
http://www.parc.xerox.com/spl/projects/aop/tr-d.htm